

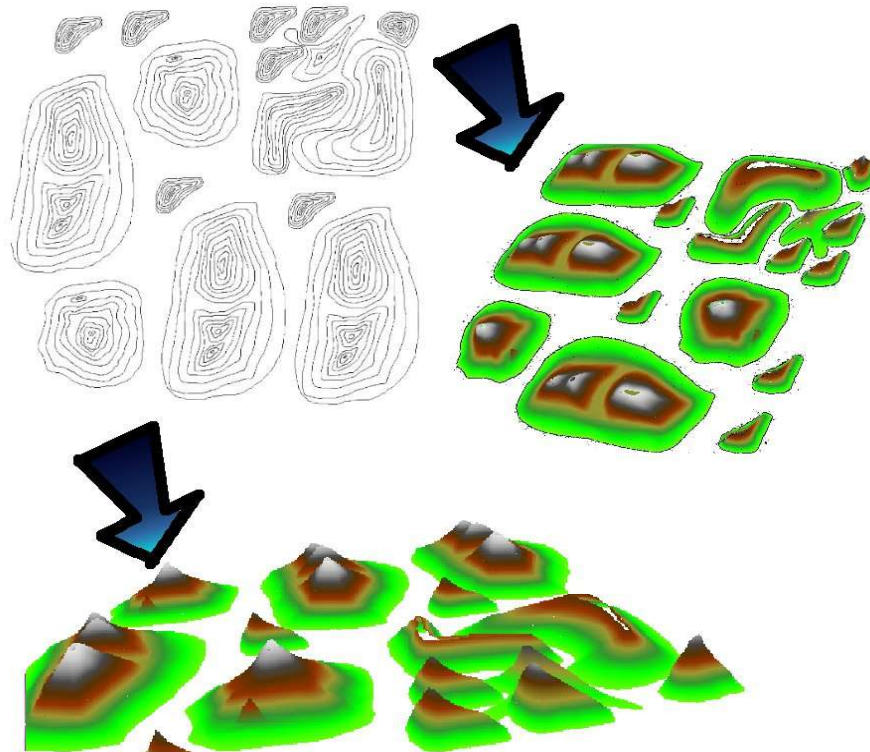
Progetto per il corso di Algoritmi e Strutture dati.

Nicola Mosca

matricola 104818

Titolo del progetto:

# GM generatore di montagne tridimensionali



- Introduzione
  - descrizione veloce del progetto
- analisi progetto
  - descrizione dettagliata funzionamento
  - descrizione e motivazione strutture dati utilizzate
    - Image
    - Pixel
    - Cartina
    - Punto
    - Isoipsa
    - Array\_punti
    - Array\_isoipse
    - Lisa\_iso
  - descrizione e motivazione algoritmi utilizzati
    - la creazione delle isoipse partendo dal file BMP
    - la creazione delle relazioni tra le isoipse
      - versione 1 (lista di relazioni)
      - versione 2 (foresta di relazioni)
    - la determinazione dell'altezza delle isoipse
      - versione 1 (lista di relazioni)
      - versione 2 (foresta di relazioni)
    - la rappresentazione della scena
      - versione 1 (lista di relazioni)
      - versione 2 (foresta di relazioni)
  - discussione su efficienza algoritmi e possibili miglioramenti
- documentazione progetto
  - istruzioni per la compilazione
  - istruzioni per l'utilizzo

## Introduzione

### Descrizione veloce del progetto

Il progetto è un generatore di montagne tridimensionali, per il funzionamento necessita di un file in formato bmp con le curve di livello (isoipse) della montagna (o delle montagne) che si vogliono generare. Il software legge il file di input e genera la scena tridimensionale (utilizzando le funzioni delle librerie grafiche OpenGL) è poi possibile “navigare” nella scena creata utilizzando la tastiera.

## Analisi del progetto

### Descrizione dettagliata funzionamento

Il programma prende i dati in ingresso da linea di comando, nel caso in cui venga passato un numero di parametri errato il programma termina segnalando all'utente la chiamata corretta da effettuare.

L'unico parametro da passare è in path del file .bmp che contiene le isoipse della scena da rappresentare, inizialmente si verifica l'esistenza del file e la sua correttezza (formato BMP, larghezza uguale all'altezza, bpb (i byte per pixel devono essere 24), ecc) questi controlli vengono fatti leggendo l'header del file. Una volta letto il tutto si procede ad allocare la memoria necessaria a contenere tutti i dati dell'immagine, questi dati verranno poi interpretati, si ricreeranno tutti i pixel dell'immagine (si veda la sezione strutture dati per maggiori dettagli) e verranno salvati in una matrice bidimensionale con larghezza e altezza pari alla larghezza e altezza in pixel dell'immagine.

Quando viene creata la matrice bidimensionale di pixel si libera la memoria occupata dai dati dell'immagine.

A questo punto viene chiamata la funzione che analizzando il colore e la posizione di tutti i pixel e utilizzando l'algoritmo di ricostruzione della isoipsa, ricrea tutte le isoipse salvandole in una apposita struttura dati dinamica.

Quando tutte le isoipse sono state create la memoria occupata dalla matrice bidimensionale di pixel viene liberata.

Viene chiamata una funzione che confronta ogni isoipsa con le altre, per capire quali sono le relazioni (una isoipsa può avere più figli ma un solo padre) le relazioni vengono salvate in una lista di "Relazioni" ogni elemento Relazione contiene un puntatore alla isoipsa bassa e un puntatore all'isoipsa alta.

Parallelamente alla creazione delle relazioni viene fatto anche il calcolo dell'altezza di ogni isoipsa (più un'isoipsa è contenuta in altre isoipse più è alta)

- Infine si giunge alla rappresentazione grafica della scena, che si ottiene stampando tutte le relazioni create (in dettaglio):
  - si assume che i punti costituenti l'isoipsa alta sono minori di quelli dell'isoipsa bassa
  - la montagna è creata da tutte le "fasce" da una isoipsa bassa ad una isoipsa alta
  - ogni fascia è creata facendo un trapezio da 2 punti dell'isoipsa bassa a 2 punti dell'isoipsa alta.
  - Dato che i punti dell'isoipsa bassa sono maggiori dei punti dell'isoipsa alta per calcolare i vari punti che formeranno i vari trapezi si segue questa regola:

$\text{rapporto} = \text{punti\_isoipsa\_bassa} / \text{punti\_isoipsa\_alta}$

$\text{punti isoipsa alta} = x \text{ e } x+1$

$\text{punti isoipsa bassa} = x * \text{rapporto} \text{ e } x * \text{rapporto} + \text{rapporto}$

## Descrizione e motivazione strutture dati utilizzate

viene fatto ora l'elenco delle strutture dati utilizzate, corredato da motivazioni relative alle scelte fatte.

```
typedef struct Image{  
  
    unsigned long sizeX;//larghezza in pixel dell'immagine  
    unsigned long sizeY;//altezza in pixel dell'immagine  
    char *data;//contenuto dell'immagine  
  
}Image;
```

Questa è la struttura dati che viene creata interpretando il file bitmap, la larghezza e l'altezza vengono lette dalle intestazioni header del file, mentre i dati dell'immagine (i colori di tutti i pixel) saranno contenuti in un array di char. Si è utilizzato un array di char per il seguente motivo: l'immagine è una bitmap a 24 bpb, ciò significa che per ogni pixel ci saranno 3 byte di informazioni

1 byte = 8 bit (Red)  
1 byte = 8 bit (Green)  
1 byte = 8 bit (Blue)

la dimensione dell'array data sarà quindi pari a  $sizeX * sizeY * 3$  (perché ogni pixel necessita di 3 byte)

Per una più comoda gestione viene fatta una struttura anche per il colore:

```
typedef struct Colore{  
  
    int r;//componente rossa da 0 a 255  
    int g;//componente verde da 0 a 255  
    int b;//componente blu da 0 a 255  
  
}Colore;
```

Tutti i dati contenuti nella struttura Image, verranno analizzati e salvati in una struttura Cartina, sotto forma di matrice bidimensionale di Pixel

```
typedef struct Pixel{  
  
    int x;  
    int y;  
    Colore c;  
    int stato;//0=libero 1=usato  
}Pixel;  
  
typedef struct Cartina{  
  
    Pixel *dati;
```

```
int width;//larghezza in pixel
int height;//altezza in pixel
```

```
}Cartina;
```

Per un comodo accesso alla matrice bidimensionale dei Pixel allocata dinamicamente sono state create delle funzioni tipo

```
Pixel* get_punto(Cartina *,int,int);
```

Che ritornano il puntatore al pixel presente nella Cartina passata, avente coordinate x e y (la funzione è necessaria perchè la matrice è allocata dinamicamente, e quindi non si ha accesso al singolo elemento usando i 2 indici tipo  $v[i][j]$  ma bisogna fare  $v[j*\text{larghezza}+i]$ )

Passiamo ora alle strutture date “importanti”

```
typedef struct Punto{
    int x;
    int y;
}Punto;
```

Identifica un punto, con le due coordinate, il colore non serve più perchè i punti saranno legati ad una isoipsa.  
Ogni isoipsa infatti avrà al suo interno una struttura dati di tipo Array\_punti

```
typedef struct Array_punti{
    int initialized;//variabile di controllo
    float growth;//crescita in percentuale
    int size;//dimensione effettiva
    int indice;//indice di inserimento
    Punto *data;//array di elementi
}Array_punti;
```

initialized - serve per capire se la struttura è già stata inizializzata

growth – specifica di quanto deve crescere l'array data ogni volta che raggiunge la dimensione limite

size – la dimensione limite

indice – indica lo stato attuale di riempimento della struttura

\*data – è l'array di punti, quello che contiene i dati

Array\_punti è una struttura particolare, simile ad uno Stack , questa struttura viene inizializzata con la funzione

```
void initialize_Array_punti(Array_punti *,int,float);
```

che specifica la dimensione iniziale e la percentuale di crescita della struttura

Ogni volta che si inserisce un nuovo punto viene chiamata la funzione

```
void add_element_Array_punti(Array_punti *);
```

che verifica lo stato di indice, e se questo è uguale a size, procede con la funzione

```
void cresci_Array_punti(Array_punti *);
```

fondamentale in questa funzione è l'utilizzo di

```
void * realloc(void*,int);
```

questa funzione prende come parametri un puntatore precedentemente allocato e la nuova dimensione da allocare, ritorna il puntatore al nuovo array allocato; se la dimensione da allocare è maggiore di quella creata precedentemente lo spazio in più verrà allocato ma non verrà inizializzato (in sostanza le variabili in quella memoria conterranno “sporcizia”), mentre se la nuova dimensione da allocare è minore rispetto a quella vecchia lo spazio vecchio in più verrà semplicemente reso disponibile.

Motivazioni di questa scelta

Il motivo per cui si è adottata una soluzione di questo tipo è dettato dal fatto che i punti vengono letti già nell'ordine corretto dall'algoritmo di ricostruzione dell'isoipsa (che verrà spiegato più avanti), in sostanza serviva una struttura che non sprechi troppa memoria e che permetta un rapido accesso ai dati (con l'indice dell'array riesco ad accedere con tempo  $O(1)$  )

Passiamo ora alla definizione della struttura Isoipsa

```

typedef struct Isoipsa{

    Array_punti punti;//punti che formano l'isoipsa

    int altezza;//altezza dell'isoipsa

    //queste variabili servono per calcolare la disposizione delle isoipse
    int min_x,max_x,min_y,max_y;

}Isoipsa;

```

Isoipsa contiene una struttura Array\_punti, questi saranno i punti che formano il contorno dell'isoipsa, altezza definisce l'altezza dell'isoipsa, le variabili min\_x,max\_x,min\_y,max\_y servono per verificare le relazioni tra le isoipse.

Le isoipse verranno salvate in una struttura dati di tipo

```

typedef struct Array_iso{

    int initialized;//variabile di controllo

    float growth;//crescita in percentuale
    int size;//dimensione effettiva
    int indice;//indice di inserimento

    Isoipsa *data;//array di elementi

}Array_iso;

```

Questa struttura ha le stesse caratteristiche di Array\_punti, non verrà quindi commentata in dettaglio.

L'ultima struttura da analizzare è la lista di relazioni tra isoipse, che può essere vista anche come un grafo.

```

typedef struct Relazione{

    Isoipsa* basso;//indice dell'isoipsa in basso
    Isoipsa* alto;//indice dell'isoipsa in alto

    struct Relazione *next;//puntatore al prossimo elemento
}Relazione;

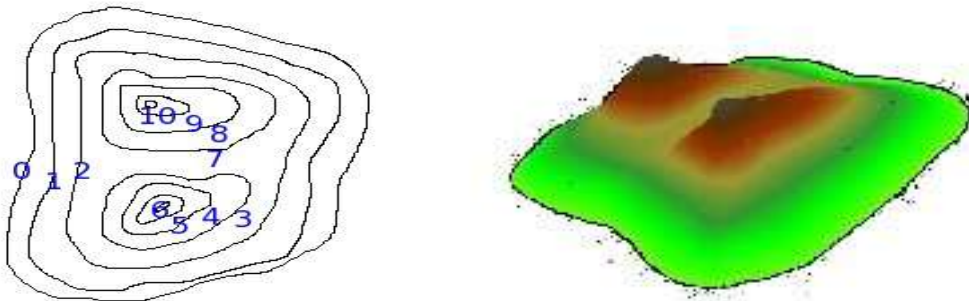
```

```
typedef struct Dummy{
```

```
    Relazione *first;  
    Relazione *last;  
}Dummy;
```

Dummy è la sentinella, che contiene un puntatore al primo e all'ultimo elemento della lista.

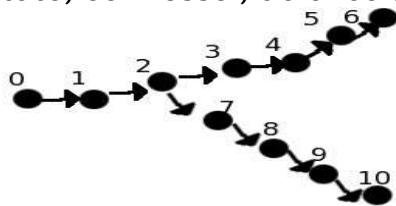
Faccio ora un esempio di come viene interpretata l'immagine di sinistra (senza numeri), che genera l'immagine di destra



se stampiamo la lista delle relazioni di questo caso (per ogni relazione viene stampato l'indice dell'isoipsa bassa e dell'isoipsa alta) otteniamo il seguente output:

```
vado da 9 a 10  
vado da 8 a 9  
vado da 7 a 8  
vado da 2 a 7  
vado da 5 a 6  
vado da 4 a 5  
vado da 3 a 4  
vado da 2 a 3  
vado da 1 a 2  
vado da 0 a 1
```

che può essere rappresentato anche con il seguente grafo non orientato, connesso , aciclico (un albero) l'intera scena sarà una foresta



Questo grafo può essere utilizzato per calcolare l'altezza di ogni isoipsa.

## Descrizione e motivazione algoritmi utilizzati

Si passa ora alla descrizione degli algoritmi utilizzati nel progetto, si possono isolare nel progetto analizzato 3 problemi principali

- la creazione delle isoipse partendo dal file BMP
- la creazione delle relazioni tra le isoipse
- la determinazione dell'altezza delle isoipse
- la rappresentazione della scena

## La creazione delle isoipse partendo dal file BMP

Viene scandita la matrice bidimensionale dei pixel dell'immagine, salvata nella struttura Cartina, ogni volta che si trova un pixel di colore nero (volendo si può agevolmente cambiare il colore desiderato) si aggiunge una nuova isoipsa nell'array Array\_iso, a questo punto viene chiamata la funzione

```
void add_punto(Cartina *cart, Colore *col, Isoipsa *iso, int x, int y)
```

questa funzione è la chiave di questo algoritmo e fa le seguenti operazioni:

- segna il pixel nella Cart con coordinate x,y come letto
- aggiunge un nuovo punto nell'array\_punti di iso con le coordinate x,y
- controlla gli 8 pixel che circondano il pixel corrente

x-1,y+1	x,y+1	x+1,y+1
x-1,y	x,y	x+1,y
x-1,y-1	x,y-1	x+1,y-1

Se trova un pixel che ha lo stesso colore di col e che non è ancora stato letto, chiama se stessa passando le coordinate del nuovo punto.

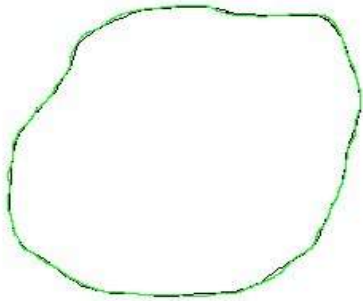
Questa procedura può essere applicata ad un qualsiasi problema dove sia necessario seguire una linea.

NOTA IMPORTANTE la funzione non legge tutti i pixel che circondano il pixel letto al momento (quello con coordinate  $x,y$ ). Ciò che succede in un caso standard è questo:

l'algoritmo parte e comincia ad inserire i punti dell'isoipsa nella struttura dati apposita, quando l'isoipsa verrà completata la condizione del punto adiacente a quello appena letto non verrà più rispettata (perché anche se il colore è nero il pixel sarà stato segnato come letto) a questo punto la ricorsione continua e, se non ci fosse un controllo i punti lasciati indietro (perché in abbondanza) verrebbero aggiunti come punti finali dell'isoipsa (ma ciò è sbagliato) per evitare questo si è operato nel seguente modo:

- il punto viene segnato sempre e comunque come letto
- il punto viene aggiunto all'isoipsa se e solo se la differenza delle sue coordinate  $x,y$  non è maggiore di 1 rispetto alle coordinate  $x,y$  dell'ultimo punto inserito

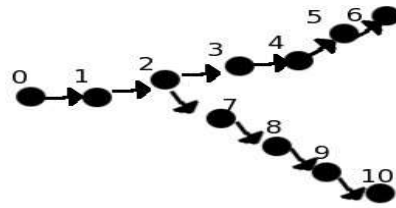
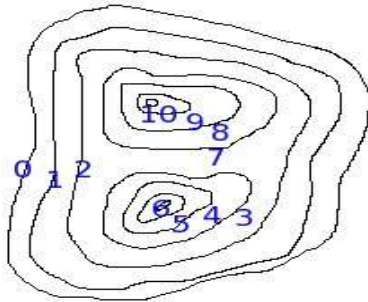
viene fatta ora una figura per chiarire maggiormente la situazione:



nell'immagine si vede una linea nera, i pixel originali, e una linea verde, i pixel letti, i rimasugli neri non ancora letti (perché erano in eccesso) non devono essere considerati come punti dell'isoipsa (perché l'ho già costruita) ma devono comunque essere segnati come letti, altrimenti potrebbero essere interpretati come l'inizio di una nuova isoipsa.

## La creazione delle relazioni tra le isoipse

Questo è un'altro algoritmo molto importante, per il quale si è dovuti ricorrere ad una approssimazione che non ne garantisce il funzionamento corretto nel 100% dei casi, in sostanza quello che deve fare questo algoritmo è verificare se una isoipsa è contenuta o no in un'altra, questo serve a definire l'insieme delle relazioni tra isoipse al fine di generare la scena completa, ad esempio avendo questa cartina viene generata la seguente lista di relazioni



l'isoipsa 0 contiene la 1  
 l'isoipsa 1 contiene la 2  
 l'isoipsa 2 contiene la 3  
 l'isoipsa 2 contiene la 7  
 ecc...

come si vede una isoipsa può contenere più isoipse

per vedere se l'isoipsa  $i$ -esima contiene la  $j$ -esima, vengono calcolati i seguenti valori

- minima coordinata  $x$  dei punti dell'isoipsa
- massima coordinata  $x$  dei punti dell'isoipsa
- minima coordinata  $y$  dei punti dell'isoipsa
- massima coordinata  $y$  dei punti dell'isoipsa

poi vengono confrontati questi valori con un'altra isoipsa e se i massimi di  $I$  sono maggiori dei massimi di  $J$  e i minimi di  $I$  sono minori dei minimi di  $J$  allora  $I$  contiene  $J$ .

Questo controllo non funziona sempre, ma da un risultato accettabile se le scene che si vanno a rappresentare non sono troppo complesse.

NOTA:

il calcolo del minimo e del massimo per ogni isoipsa viene fatto scandendo l'array dei punti dell'isoipsa e memorizzandone i valori massimi e minimi.

Si è scelta questa operazione perchè permette di calcolare tutti e 4 i valori con costo  $O(n)$ .

Avrei potuto optare anche per algoritmi di ordinamento più efficienti come QuickSort, MergeSort o HeapSort, ordinare l'array e poi prendere l'indice iniziale e quello finale (il minimo e il massimo). Il problema è che i dati sono già ordinati secondo un preciso criterio, quindi avrei

dovuto copiare i dati in un array temporaneo, e già qui ho un costo  $O(n)$ , poi fare 2 ordinamenti (uno per la  $x$  e uno per la  $y$ ), infine prendere i valori minimi e massimi. La seconda soluzione è molto più onerosa, sia in termini di tempo che di memoria, quindi ho optato per la prima.

Le relazioni possono essere salvate in due modi (vedere i due parametri differenti che si possono passare al makefile)

- in una lista di Relazioni  
viene fatta una semplice lista di relazioni, ogni elemento della lista contiene un puntatore all'isoipsa alta e bassa  
Per costruire la lista si parte dall'ultima isoipsa letta e la si confronta utilizzando i massimi e i minimi con tutte le altre isoipse lette prima di lei, appena si trova una isoipsa che la contiene viene aggiunto l'elemento alla lista con in 2 puntatori, e si passa all'isoipsa penultima...avanti così fino alla prima
- in una foresta di Relazioni  
per ogni isoipsa si crea una lista, e all'interno di questa lista vengono messe tante relazioni quante sono le isoipse contenute dall'isoipsa che si sta analizzando

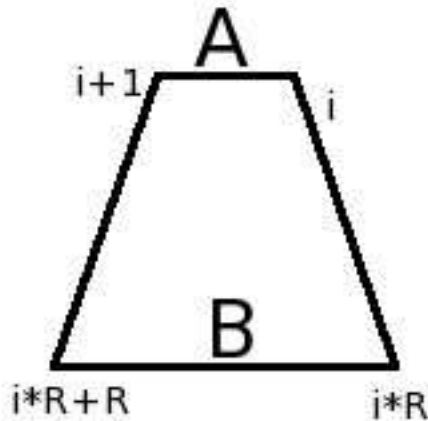
## La determinazione dell'altezza delle isoipse

- in una lista di Relazioni  
anche se si potrebbe usare la lista di relazioni per calcolare l'altezza di una isoipsa, la soluzione più semplice è quella di partire dalla prima isoipsa letta, e riconfrontarla con tutte le altre isoipse, ogni volta che una isoipsa risulta contenuta in un'altra viene incrementata la sua altezza di 1  $O(n^2)$  dove  $n$  è il numero di isoipse
- in una foresta di Relazioni  
viene scandito l'array delle liste, e si incrementa di 1 l'altezza dell'isoipsa che figura come alta in ogni elemento Relazione

## La rappresentazione della scena

Il principio di rappresentazione è uguale, sia in un caso che nell'altro a parte alcune particolarità che verranno spiegate in seguito, in sostanza la rappresentazione consiste nel generare tutte le "fasce" che vanno da una isoipsa a quella che segue. La fascia è costruita nel seguente modo, dato che i punti dell'isoipsa bassa sono sicuramente maggiori dell'isoipsa alta (perché la contiene) viene fatto un rapporto tra il

numero di punti dell'isoipsa bassa e quelli dell'isoipsa alta, d'ora in poi questo rapporto verrà chiamato R.  
si forma un trapezio con le seguenti caratteristiche:



vediamo che il trapezio viene costruito con 4 punti, i 2 punti della base bassa B vengono presi utilizzando l'indice corrente  $i$  \* il rapporto calcolato in precedenza, mentre i punti dell'isoipsa superiore sono semplicemente consecutivi. In realtà nel codice è presente un'ulteriore variabile che sostituisce 1, con questa variabile è possibile definire la qualità di rappresentazione.

Naturalmente il tutto è messo all'interno di un ciclo for che costruisce trapezi fino a quando non vengono stampati tutti i punti dell'isoipsa alta; ci sono anche dei controlli per evitare che gli indici vadano in overflow.

Questa è la logica di creazione delle "fasce" a questo punto possiamo analizzare i due modi differenti di rappresentare il tutto.

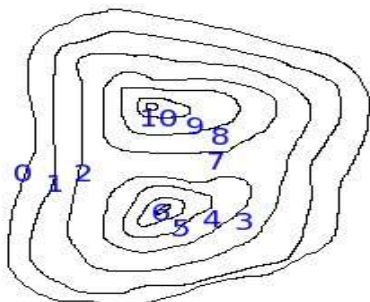
versione 1 (lista di relazioni)

Questo è il più semplice, non fa altro che visitare tutti gli elementi della lista (l'ordine non importa, tanto ogni isoipsa ha la sua altezza) e stampare una fascia per ogni elemento che trova, dall'isoipsa bassa a quella alta. A mio avviso questa è la soluzione più efficiente e stabile.

versione 2 (array di liste di relazioni)

Questa versione è un po' più complicata, nel senso che bisogna vedere quali relazioni stampare per ogni lista, ad esempio: in ogni lista figura sempre la stessa isoipsa alta e solo una isoipsa bassa.

ESEMPIO:



l'isoipsa 0 contiene la 1,2,3,....

qui andrà disegnata una sola fascia, quella da 0 a 1

andiamo all'isoipsa 2

l'isoipsa 2 contiene la 3,4,5,6,7,8,9,10

in questo caso andrà stampata la fascia tra 2 e 3 e anche quella tra 2 e 7.

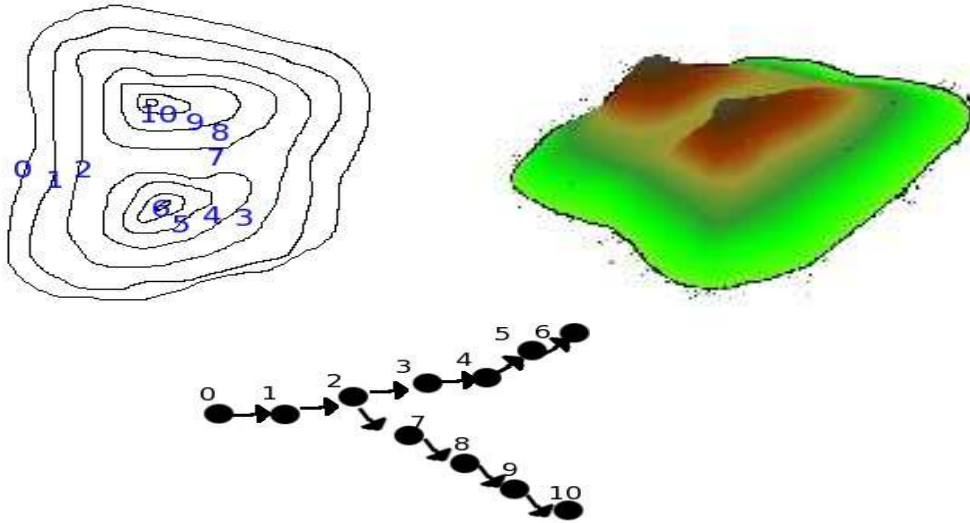
Per ovviare a questo problema io ciclo tutta la lista e stampo solo le fasce dove la differenza di altezza tra l'isoipsa bassa e quella alta è 1.

Naturalmente questo processo è più lungo, e ciò si nota anche durante la rappresentazione.

## Discussione su efficienza algoritmi e possibili miglioramenti

a parte le note già fatte in precedenza si possono fare altre notazioni:

- l'algoritmo di rilevamento dei punti dell'isoipsa, nella fase si "segnalazione dei punti non usati" a volte inserisce un punto anche quando l'isoipsa è già stata completata, ciò accade quando viene a trovarsi un punto in eccesso adiacente all'isoipsa con x o y uguali all'ultimo punto inserito.
- Relativamente al controllo sulle relazioni delle isoipse, a volte questo controlla ritorna dei valori falsi, in pratica quando i confronti tra massimi e minimi non risulta valido. Ho pensato ad una soluzione che calcola tutti i punti dell'area dell'isoipsa, ma data la sua complessità e la mancanza di tempo materiale eventualmente questa soluzione può essere discussa in sede d'esame.
- Relativamente alla rappresentazione di più isoipse con la stessa isoipsa bassa, la soluzione che ho adottato è una approssimazione, infatti se analizziamo il solito esempio



vediamo che l'isoipsa 2 ha due isoipse alte, la 3 e la 7, in questa situazione vengono generate due fasce, una che va da 2 a 3 e una che va da 2 a 7, in questo caso il risultato che si ottiene è buono, in altri casi dove nell'isoipsa 2 ci sono particolari concavità, soprattutto nella zona centrale, tra l'isoipsa 3 e la 7, alcune concavità potrebbero essere "scavalcate" dalla fascia che va da 2 a 3 o quella che va da 2 a 7.

- la "punta della montagna" se non è disegnata, nel file di origine non viene creata, anche per questa ho una soluzione di cui eventualmente sene può discutere, in sostanza basta fare una Relazione che abbia come isoipsa bassa l'ultima di quella montagna e come isoipsa alta un unico punto alla stessa altezza.

Il risultato che si ottiene nella generalità dei casi credo sia comunque accettabile.

## Istruzioni per la compilazione

Per compilare il seguente progetto è necessario avere:

- compilatore gcc (io ho utilizzato la versione 3.3.4)
- librerie grafiche OpenGL - [www.opengl.org](http://www.opengl.org)
- librerie grafiche libglut - [http://www.opengl.org/resources/libraries/glut/glut\\_downloads.html](http://www.opengl.org/resources/libraries/glut/glut_downloads.html)
- comando makefile

è sufficiente assicurarsi della presenza del makefile e digitare il comando make, che produrrà il seguente output

```
make
# make mg_list  crea l'eseguibile con la lista di relazioni
# make mg_arr   crea l'eseguibile con l'array di liste di relazioni
# make clean    pulisce i file oggetto e elimina gli eseguibili
```

come si può leggere passando diversi parametri a make è possibile compilare una delle due versioni implementate, oppure eliminare i file oggetto e gli eseguibili per una nuova compilazione.

## Istruzioni per l'utilizzo

- Si consiglia di eseguire il programma su una macchina con accelerazione grafica 3D
- Il file di origine deve essere in formato bitmap (.bmp)
- il file di origine deve avere altezza e larghezza uguali
- le isoipse devono essere di colore nero (0x000000) (volendo è possibile cambiare il colore, nei due file mg\_arr.c o mg\_list.c)
- le dimensioni del file possono andare da 256 pixel a 2500 pixel (si consiglia di non andare oltre i 2000 pixel, perchè durante la rotazione della scena in alcune zone i bordi possono uscire dalla rappresentazione, questo è comunque un problema che si può risolvere aumentando l'area di disegno con le openGL)
- si consiglia di utilizzare linee con larghezza di un pixel per delimitare le isoipse
- il programma viene chiamato passando il path del file (relativo o assoluto) come unico parametro
- nella cartella bin sono presente alcune immagini con cui fare degli esempi
- è possibile "navigare" nella scena utilizzando i tasti a,d,w,z,q,e
  - a: sposta la scena a sinistra
  - d: sposta la scena a destra
  - w: si allontana dalla scena
  - z: si avvicina alla scena
  - q: alza la scena
  - e: abbassa la scena

In caso di problemi, necessità chiarimenti non esitate a contattarmi

Nicola Mosca  
matricola 104818

[info@chiese.tn.it](mailto:info@chiese.tn.it)  
[nik600@hotmail.com](mailto:nik600@hotmail.com)  
[nicola.mosca@lavalleinvisibile.net](mailto:nicola.mosca@lavalleinvisibile.net)